An Efficient Pattern Matching Approach for Compressed Genomic Sequences

Han Duan*†, Chuan Hu*†, Zhihong Shen*
*Computer Network Information Center, Chinese Academy of Sciences, Beijing, China
†University of Chinese Academy of Sciences, Beijing, China
E-mail: {hduan, huchuan, bluejoe}@cnic.cn

2025 IEEE International Conference on Bioinformatics and Biomedicine (BIBM) | 979-8-3315-1557-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/BIBM66473.2025.11356792

***Abstract*—The rapid growth of genomic data has created a demand for storage-efficient formats that also support fast search. Most existing compressors, whether general-purpose or domain-specific, require full decompression before analysis, resulting in substantial time and memory costs. We present an efficient pattern matching approach for compressed genomic sequences, combining a *Gene Sequence Bitmap Compression* (GSBC) scheme, a *Low-Frequency Byte Position-Based Pattern Matching* (LFBPBPM) algorithm, and an optimized indexing method. GSBC applies a 2-bit nucleotide encoding to reduce storage while preserving biological semantics. LFBPBPM exploits the skewed byte-frequency distribution in compressed sequences for efficient candidate filtering, avoiding decompression. The indexing strategy, based on *Sequence Indexing Triads* and horizontal compression, further reduces index size by up to 28%. Experiments on large genomic datasets show that our method achieves a 29% compression ratio with lower memory usage and faster matching than 18 compressors and 11 search algorithms, offering a promising solution for large-scale genomic data storage and retrieval.**

***Index Terms*—bioinformatics, sequence compression, compressed pattern matching**

I. Introduction

Advances in high-throughput sequencing have driven explosive growth in genomic data, with modern reference genomes reaching gigabase scales and population-scale projects generating terabytes daily [1]. Efficient storage and rapid retrieval of such data are essential for downstream applications in personalized medicine, comparative genomics, and genome-wide association studies (GWAS) [2].

Conventional storage formats(e.g.FASTA/FASTQ [3]) and general-purpose compressors (e.g.gzip [4],bzip2 [5]) reduce file size but require full decompression before analysis, incurring substantial time and memory overhead. Domain-specific compressors such as DSRC [6], GTZ [7] and NAF [8] exploit sequence redundancy to improve compression ratios, yet still depend on decompression for querying, limiting scalability for large datasets.

Compressed-domain pattern matching has shown promise in text and image data, but its application in genomics remains limited [9]. Most existing genomic search methods operate on decompressed sequences or maintain large uncompressed indexes, both of which are inefficient for repeated queries [10]. The distinctive statistical characteristics of genomic sequences present untapped potential for specialized compressed-domain algorithms.

This work presents an efficient pattern matching approach for compressed genomic sequences that combines:

- **Gene Sequence Bitmap Compression (GSBC):** A 2-bit nucleotide encoding that reduces storage while preserving biological semantics and enabling direct compressed-domain computation;
- **Low-Frequency Byte Position-Based Pattern Matching (LFBPBPM):** A search method leveraging skewed byte-frequency distributions for sublinear-time matching without decompression;
- **Index Optimization:** A compact indexing strategy using Sequence Indexing Triads (SITs) and horizontal compression, reducing index size without compromising query speed.

Extensive experiments on large genomic datasets demonstrate that our approach achieves lower memory usage, faster search, and competitive compression ratios compared with 18 compressors and 11 search algorithms, providing a promising direction for scalable genomic data storage and retrieval.

II. Related Work

Existing research related to our work can be categorized into three main areas: genomic sequence compression, pattern matching algorithms, and compressed-domain indexing strategies.

A. Genomic Sequence Compression

General-purpose compressors such as `gzip`, `bzip2`, and `brotli` [11] are widely used for genomic data storage due to their ease of deployment. However, they fail to exploit the inherent redundancy and small alphabet size of DNA sequences, leading to suboptimal compression ratios. Domain-specific compressors such as DSRC, GTZ and NAF achieve better compression by leveraging reference-based encoding, quality score modeling, and run-length encoding tailored to genomic data. Nevertheless, these methods still require full decompression before search operations, which limits their use in real-time analysis pipelines.

B. Pattern Matching Algorithms

Classical pattern matching algorithms, including Boyer–Moore(BM) [12], Quick Search(QS) and Zhu-Takaoka(ZT) [13] operate efficiently on uncompressed sequences but are inapplicable to compressed-domain search.

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on August 17,2026 at 02:13:33 UTC from IEEE Xplore. Restrictions apply.

More advanced methods such as Knuth–Morris–Pratt (KMP), First-Last Pattern Matching(FLPM) and Processor-Aware Pattern Matching(PAPAM) [14] have improved search performance by optimizing filtering and verification stages, yet they still operate in the uncompressed domain. In contrast, our LFBPBPM algorithm exploits low-frequency byte anchoring directly in the GSBC-compressed space, significantly reducing candidate matches and avoiding decompression overhead.

C. Compressed-Domain Indexing

Index structures for genomic search, such as FM-index [15] and suffix arrays [16], have been adapted to compressed data in text retrieval. However, their application in genomics often results in large index sizes or complex construction processes, especially when dealing with gigabase-scale datasets [17]. Few works have addressed index optimization for compressed genomic sequences. Our indexing approach—based on Sequence Indexing Triads (SITs), All Index Tables (AITs), and horizontal compression—reduces storage while retaining fast access to candidate positions, enabling efficient large-scale genomic search.

In summary,while prior studies have achieved significant advances in either compression efficiency or search performance, they rarely optimize both simultaneously in the genomic context. Our work aims to address this gap by integrating an efficient compression scheme, a decompression-free pattern matching algorithm, and a compact indexing structure, providing an effective solution for optimizing pattern matching in large-scale genomic data.

III. Fundamentals of Genomic Bit Mapping

This section outlines the fundamentals of gene sequence compression and pattern matching, along with key concepts and notations forming the basis for subsequent algorithm design and analysis.

A. Compressed Pattern Encoding

Let Σ be an alphabet with size σ. A string is a sequence of characters drawn from Σ, arranged in a specific order. The length of a string S is denoted by $|S|$, and $S[i]$ represents the i-th character of S (where $0 \le i \le |S|-1$). A substring of S from position i to j is denoted by $S[i]S[i+1]\dots S[j]$, where $0 \le i \le j \le |S|-1$.

The classical string matching problem is defined as follows: Given a text string T of length n and a pattern string P of length m, the goal is to find all positions i such that:

$$T[i]T[i+1]\dots T[i+m-1] = P, \quad 0 \le i \le n-m. \quad (1)$$

This work focuses on the pattern matching problem for gene sequences, where the alphabet $\Sigma = \{\text{A}, \text{G}, \text{C}, \text{T (or U)}\}$, hence $\sigma = 4$. Let T be the gene sequence text and P be the gene pattern. The encoded text string is denoted by T', and its length is $n' = \lceil \frac{n}{4} \rceil$, with the unit of bytes. Given a known encoding rule E and the compressed text T', the objective is to locate all positions i in the original text T where pattern P occurs, without decompressing T'.

Considering base pairing rules (A pairs with T or U; C pairs with G), and the binary symmetry of bits in computer logic, the following mapping function(M) is defined:

$$\text{M(A)} \to 00, \quad \text{M(T(U))} \to 11, \quad \text{M(C)} \to 01, \quad \text{M(G)} \to 10.$$

B. Compressed Text Encoding

The mapping rule described above provides a binary representation of gene sequences. However, since real-world gene sequences may vary in length, padding and structural encoding mechanisms are required to support efficient matching and retrieval. We therefore define the encoded text T using a triplet form to ensure precise alignment and completeness.

Let $T = t_0 t_1 \dots t_{n-1}$ be a gene sequence. The encoded representation of T, denoted by T', is defined as a triplet:

$$T' = [T_n, T_l, \text{Mask}_T], \quad (2)$$

where:

- T_n is the **main byte sequence**, with length $n_n = \lfloor \frac{n}{4} \rfloor$, where:

$$T_n[i] = E(T[4i], T[4i{+}1], T[4i{+}2], T[4i{+}3]), \quad 0 \le i < n_n \quad (3)$$

- T_l is the **tail encoding byte**, constructed when the length n is not divisible by 4. Let $r = n \bmod 4$ with $1 \le r < 4$, then:

$$T_l = E(T[n'-r], T[n'-r+1], \dots, T[n'-1], \underbrace{A, \dots, A}_{4-r}), \quad (4)$$

  where padding with base 'A' is used to complete a full byte.
- Mask_T is the **bitmask** associated with T_l, indicating the valid bits in the final byte. The mask is a byte in which the first r bits are set to 1 and the remaining bits are set to 0:

$$\text{Mask}_T = \underbrace{11\dots1}_{r}\underbrace{00\dots0}_{4-r}. \quad (5)$$

An example of this encoding is illustrated in Table I:

TABLE I
Example of Encoded Text Representation

$\mathbf{T}$	GATT TGGG GTTC AAAG CAG
$\mathbf{T}_n$	10001111 11101010 10111101 00000010
$\mathbf{T}_l$	01001000
$\mathbf{Mask_T}$	11111100

The final encoded byte, denoted as $\mathbf{T}_l$, is obtained by appending one base 'A' to the tail sequence "CAG" to form "CAGA," which is encoded as `01001000`, where `010010` represents "CAG" and the trailing `00` corresponds to the padded 'A'. $\text{Mask}_\mathbf{T} =$ `11111100` indicates that only the first $r = 3$ base pairs (i.e., $3 \times 2 = 6$ valid bits) are meaningful in the last byte. The remaining two bits are padding and ignored during matching.

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on August 17,2026 at 02:13:33 UTC from IEEE Xplore. Restrictions apply.

IV. Pattern Matching in Compressed Genomic Sequences

This section focuses on the problem of efficient pattern matching in compressed gene sequence data. Two core design principles guide this research: (1) the development of a lossless compression scheme that preserves biological semantics while balancing compression ratio and query efficiency; and (2) the design of pattern matching algorithms that can operate directly on compressed data, thereby reducing computational overhead.

A. Gene Sequence Bitmap Compression (GSBC)

This section introduces the Gene Sequence Bitmap Compression (GSBC) algorithm, which builds upon the previously defined bit-mapping principles. The algorithm operates in two primary stages: data cleansing and byte mapping. The final output includes three parts: (i) core sequence mapping data, (ii) metadata for recovery, and (iii) supplementary information for further processing.

1) Data Cleansing: Let the raw gene sequence be denoted by $S = s_0 s_1 \dots s_{n-1}$, where $s_i \in \Sigma = \{A, T, C, G\}$,$0 \le i \le n-1$. The data cleansing process includes:

- **Case normalization**: All lowercase characters are converted to uppercase. Their positions and lengths are recorded in set $\mathcal{L} = \{(p, l) \mid$ a lowercase segment starting at position p with length $l\}$.
- **Unknown base handling**: All continuous segments of character 'N' are recorded as $\mathcal{N} = \{(q, h) \mid$ a segment of 'N' starting at q with length $h\}$.
- **Degenerate base processing**: Ambiguous bases (e.g., R, Y, S, W, K, M, etc.) are extracted and marked: $\mathcal{D} = \{(d, u) \mid u \subset$ degenerate bases, starting at position $d\}$.

After processing, all lowercase and degenerate bases are removed, yielding a purified sequence $S' = s'_0 s'_1 \dots s'_{n'-1}$ with $s'_i \in \Sigma$.

2) Byte Mapping: Every four consecutive bases in the purified sequence are encoded into a single byte. Each base is first converted into its 2-bit representation according to the predefined mapping rule, and the four 2-bit codes are then concatenated to form an 8-bit value.

3) Time and Space Complexity Analysis:

a) Time Complexity:: The algorithm consists of two linear phases, each with time complexity $O(n)$:

- **Data cleansing:** Scans the original sequence once to normalize cases and mark special symbols — $O(n)$.
- **Byte mapping:** Applies bitwise operations per nucleotide group to compress the data — $O(n)$.

Combining the time complexities of these two phases:

$$T(n) = O(n) + O(n) = O(n) \tag{6}$$

b) Space Complexity:: The total space consumption includes:

- **Main compressed data:** Each 4 nucleotides form 1 byte $\Rightarrow O(n/4) = O(n)$.
- **Metadata:** Positional records $\mathcal{L}, \mathcal{N}, \mathcal{D}$ are small and scale sublinearly, typically $O(1)$ relative to n.

Thus, the overall space complexity is:

$$S(n) = O(n/4) + O(1) = O(n) \tag{7}$$

B. Low-Frequency Byte Position-Based Pattern Matching(LFBPBPM)

The Low-Frequency Byte Position-Based Pattern Matching Algorithm (LFBPBPM) is a novel compressed-domain method specifically designed for matching tasks in genomic sequences. It consists of two main phases: (1) filtering candidate locations using statistically rare bytes, and (2) performing multilevel matching verification.

1) Low-Frequency Byte Skipping: Let the compressed sequence be denoted as $T' = t'_0 t'_1 \dots t'_{n'-1}$, and let the pattern set be $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$. The algorithm operates as follows:

1) **Dynamic Low-Frequency Anchor Selection:** A frequency function $\mathcal{F}_{T'} : \Sigma \to \mathbb{N}$ is defined over the alphabet Σ to capture byte occurrences in T':

$$\mathcal{F}_{T'}(\tau) = |\{i \mid T'[i] = \tau,\ 0 \le i < n'\}| \tag{8}$$

For the main encoded section P_k^m of each pattern P_k, the byte with the lowest frequency is selected as the anchor:

$$\tau_k^* = \arg\min_{\tau \in P_k^m} \mathcal{F}_{T'}(\tau) \tag{9}$$

2) **Bidirectional Verification:** For anchor byte τ_k^* at offset $\text{pos}(\tau_k^*)$ in P_k^m, we define:
   - **Forward Verification:** Validate $T'[i - \text{pos}(\tau_k^*) + j] = P_k^m[j]$ for $0 \le j \le \text{pos}(\tau_k^*)$.
   - **Backward Verification:** Validate $T'[i + j] = P_k^m[\text{pos}(\tau_k^*) + j]$ for $1 \le j \le m_k^m - \text{pos}(\tau_k^*) - 1$.

3) **Boundary Check:** Boundary bits are validated via masked bit operations:

$$\begin{cases} (T'[i_{\text{start}}] \& \text{Mask}_k^f) = P_k^f \\ (T'[i_{\text{end}}] \& \text{Mask}_k^l) = P_k^l \end{cases} \tag{10}$$

where $i_{\text{start}} = i - \text{pos}(\tau_k^*)$ and $i_{\text{end}} = i + (m_k^m - \text{pos}(\tau_k^*) - 1)$.

2) Algorithm Illustration:

a) Pattern Preprocessing: Assume the encoded pattern is $P_0 = [8, 3, 18, 8, 24]$. This pattern is decomposed into three components:

- Prefix byte: $P_0^f = 8$
- Main byte sequence: $P_0^m = [3, 18, 8]$
- Suffix byte: $P_0^l = 24$

b) Dynamic Low-Frequency Anchor Selection: As illustrated in Fig. 1(b), we consider the case where $P_0^m[1] = 18$. The frequency of this byte in the compressed text T' is $\mathcal{F}'(18) = 2$, indicating that byte 18 has the lowest occurrence among all bytes in the pattern segment. Consequently, the subsequent pattern matching process adopts byte value $b = 18$ as the anchoring criterion to filter candidate positions. After scanning the compressed sequence T', the resulting candidate position set C = [9, 13].

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on August 17,2026 at 02:13:33 UTC from IEEE Xplore. Restrictions apply.

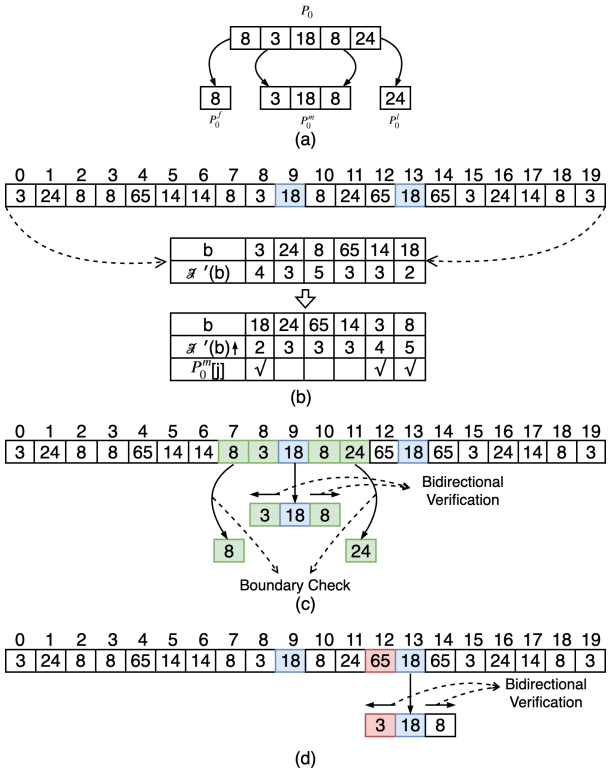

Fig. 1. Example of pattern matching using LFBPBPM algorithm

c) Bidirectional Verification and Boundary Check: In this illustrative case, the forward verification specifically targets the byte value 3, while the backward verification simultaneously targets the byte value 8.

If a candidate passes the bidirectional verification, the algorithm proceeds to boundary byte checking. Using bitwise AND and predefined masks for the prefix and suffix bytes, it validates:

$$\begin{cases} (T'[7] \& \text{Mask}_0^f) = P_0^f = 8 \\ (T'[11] \& \text{Mask}_0^l) = P_0^l = 24 \end{cases}$$

d) Failure Case: As illustrated in Fig. 1(d), the forward verification fails at candidate window position $i = 12$. As the forward verification fails, this candidate position is consequently eliminated from further consideration. The mismatch occurs as follows:

$$T'[12] = 65 \neq P_0^m[0] = 3$$

3) Time and Space Complexity Analysis: Let the length of the uncompressed sequence be n and the pattern length be m. After compression using 2-bit encoding, the lengths become $n' = \lceil n/4 \rceil$ and $m_k^m = \lceil m/4 \rceil$.The number of candidate positions satisfies $\alpha \leq n'$.

a) Time Complexity: The algorithm is composed of two main stages: candidate filtering and multilevel verification.

- **Anchor-Based Selction:** For each pattern P_k, the algorithm scans the entire compressed sequence T' to locate all occurrences of the selected low-frequency byte τ_k^*. This operation has linear time complexity $O(n)$.
- **Verification Stage:** Let α denote the number of candidate windows produced for a pattern. Each candidate requires up to $O(m')$ operations for full forward, backward, and boundary checking. Hence, the overall verification cost is $O(\alpha \cdot m')$.

Combining both components, the total time complexity is:

$$T = O(4n') + O\left(\sum_{k=0}^{3} \alpha_k \cdot m_k^m\right) = O(n + \alpha m) \quad (11)$$

- **Worst-case scenario:** When all bytes appear in the candidate list, the number of candidates reaches $\alpha = O(n') = O(n)$.
- **Average-case scenario:** Assuming a uniform distribution of byte values, the expected number of candidates is $\alpha = O(n/2^8) = O(n/256)$.

In typical biological datasets, the byte distribution is highly skewed, allowing $\alpha \ll n$. Therefore, the average-case complexity is significantly lower than the worst case.

b) Space Complexity: The space cost includes the following components:

- **Compressed Text:** The compressed sequence T' requires $O(n/4)$ space.
- **Pattern Set:** Pattern variants P occupy $O(m/4)$ space.
- **Candidate and Result Sets:** The candidate windows $\mathcal{C}[k]$ and match positions $\mathcal{R}$ require $O(n/4) = O(\alpha)$ space.

Therefore, the overall space complexity is:

$$S = O(n/4) + O(m/4) + O(\alpha) = O(n + m + \alpha) \quad (12)$$

V. Optimized Indexing for LFBPBPM

This section introduces an index structure specifically designed to optimize the LFBPBPM algorithm. The index design follows two principles: (1) efficiently locating anchor bytes to narrow the candidate search space; and (2) employing a compact representation to optimize storage, memory loading, and query speed.

A. Frequency Table Construction

The frequency table is formally defined as:

$$\mathcal{F} = \{\langle b_i,\ f_i \rangle\} \quad (13)$$

where b_i denotes the i-th byte and f_i represents its corresponding frequency. The index i satisfies $0 \leq i \leq 255$. The size of the frequency table depends solely on the byte set cardinality and is thus independent of the encoded text length.

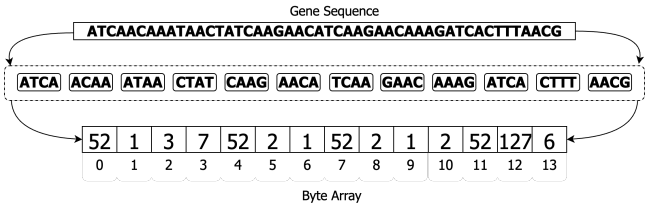

Fig. 2. Gene sequence mapping example

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on August 17,2026 at 02:13:33 UTC from IEEE Xplore. Restrictions apply.

As illustrated in Fig. 3, the frequency table is sorted in ascending order based on the frequency f_i of each byte b_i, ensuring that lower-frequency bytes are prioritized in the subsequent processing steps.

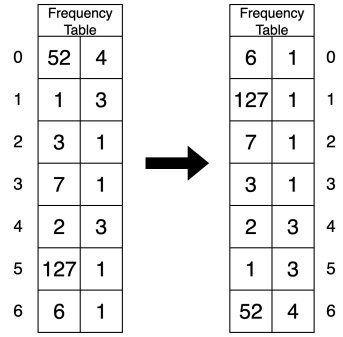

Fig. 3. Frequency table and sorted frequency table

B. Sequence Indexing Triad (SIT)

To efficiently organize and retrieve byte information within the encoded text sequence, we define a data structure called the *Sequence Indexing Triad* (SIT). Each SIT node is represented as a triplet that captures contextual and positional characteristics of a specific byte. Formally, each triplet is defined as:

$$\text{sit} = \langle p,\ h,\ d \rangle \tag{14}$$

where:

- p denotes the starting position of the current byte b in the encoded byte array;
- h is the hash value computed over the sum of byte values between the current b and the next identical byte, formally:

$$t = \sum_{i=T'[p_i+1]}^{p_{i+1}-1} b_i, \quad h = t \bmod 128 \tag{15}$$

- d represents the distance in bytes between the current b and the next occurrence of b.

As illustrated in Fig. 4, suppose $b = 52$, and one SIT entry is represented as $\langle 0,\ 11,\ 3 \rangle$.

The inclusion of the hash value h serves as a lightweight summary of the local byte context between two identical occurrences of byte b. It enables rapid filtering and comparison during pattern matching by capturing the statistical signature of the surrounding region. Moreover, in the horizontal compression introduced later, h facilitates the grouping of structurally similar triplets, thereby reducing redundancy and enhancing index compactness.

C. Index Table Row (ITR)

An *Index Table Row* (ITR) establishes a mapping from a specific byte b_i in the encoded text to its associated *Sequence Indexing Triads* (SITs). The formal definition is expressed as:

$$itr = b_i \rightarrow sit_r \tag{16}$$

As illustrated in Fig. 4, each index table row corresponds to a particular byte b_i, and the index entries r range from 0 to $f_i - 1$, where f_i denotes the frequency of byte b_i in the encoded text. Each ITR consists of two logical parts: the first part stores the byte value b_i, and the second part stores a set of SITs $\{sit_0, sit_1, \ldots, sit_{f_i-1}\}$ corresponding to all instances of b_i.

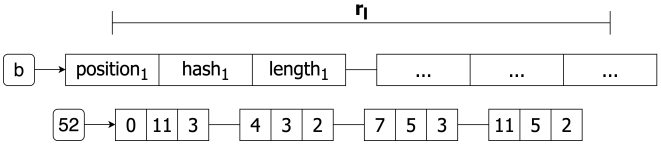

Fig. 4. Single-row index structure example

For example, suppose byte $b_i = 52$ appears four times in the encoded sequence T'. Each occurrence of the byte results in a distinct SIT entry.

D. All Index Table(AIT)

The *All Index Table* (AIT) in this work is defined as the unified collection of all index table rows ITRs, formally expressed as:

$$\mathcal{A} = \{\langle b_i,\ sit_r \rangle\} \tag{17}$$

where b_i denotes a specific byte, and sit_r represents a Sequence Indexing Triad (SIT) associated with b_i.

a) Time Complexity: The construction of the frequency table requires scanning the entire encoded sequence of length n', leading to a time complexity of O(n'). In the worst case, each byte may require scanning the entire sequence, and inserting the resulting SITs into the AIT also takes $O(n')$, yielding an overall time complexity of $O(n)$.

b) Space Complexity: The space complexity of the frequency table is O(1) since it stores at most 256 distinct byte entries, independent of the sequence length. In the worst case, each byte may appear $O(n)$ times. Thus, the AIT also has a space complexity of $O(n)$, and so does the overall system.

E. Horizontal Compression

In large-scale genomic applications, full indexing may cause excessive storage and computational overhead. To balance efficiency and key matching capability, we introduce an optimized strategy based on *horizontal compression*.

Let the set of index triples associated with a given byte in the complete index table be denoted as:

$$\mathcal{T} = \{\tau_i\}_{i=1}^{m}, \quad \text{where } \tau_i = sit_r = (p_i, h_i, d_i) \tag{18}$$

The horizontal compression consists of three main stages:

a) Preprocessing for Ordering: A strict total order is established over the set $\mathcal{T}$ using a sorting function $\varphi : \mathcal{T} \rightarrow \mathbb{R}$, which satisfies:

$$\varphi(\tau_i) > \varphi(\tau_j) \iff d_i > d_j \vee (d_i = d_j \wedge p_i < p_j) \tag{19}$$

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on August 17,2026 at 02:13:33 UTC from IEEE Xplore. Restrictions apply.

b) Length Dimension Reduction: The ordered set $\mathcal{T}'$ is partitioned into length intervals:

$$S_k = \{\tau'_i \mid d'_i = l_k\}, \quad l_k \text{ is the } k\text{-th unique distance value} \tag{20}$$

Each partition S_k undergoes the following transformation:

$$\forall \tau'_i \in S_k, \tau'_i \mapsto \begin{cases} (p'_i, h'_i, l_k) & i = \min\{j \mid \tau'_j \in S_k\} \\ (p'_i, h'_i) & \text{otherwise} \end{cases} \tag{21}$$

The first entry of each interval retains the length value l_k, while subsequent entries omit it, reducing overall storage from $O(m)$ to $O(m - n + 1)$, where n is the number of distinct length values.

c) Hash Value Aggregation: To reduce redundancy, all entries sharing the same hash value are grouped:

$$C_t = \{\tau'_i \mid h'_i = h_t\} \tag{22}$$

For each such group C_t (with $|C_t| \geq 2$), the position values are merged:

$$\bigcup_{\tau'_i \in C_t} \tau'_i \mapsto \left(\bigcup_{j=1}^{|C_t|} p'_j, h_t, l^{(t)} \right) \tag{23}$$

Here, $l^{(t)}$ represents the inherited length value. This transformation collapses multiple entries into a single record, reducing space consumption by a factor of up to $1 - \frac{1}{|C_t|}$.

d) Example Illustration: As shown in Figure 5, horizontal compression reduces four full triples into a more compact structure through length pruning and hash aggregation.

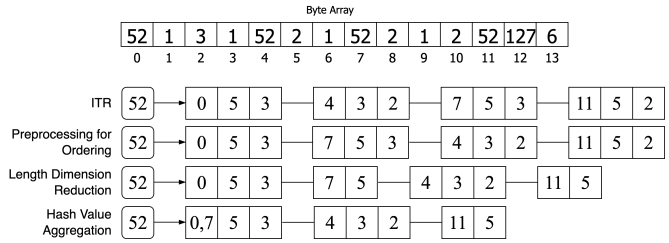

Fig. 5. Example of horizontal compression of index table

VI. EVALUATION AND DISCUSSION

A. Experimental Environment and Setup

The experiments in this study were conducted on a server running CentOS Linux 7.9, equipped with 376 GB of RAM, 220 TB of disk storage, and two Intel(R) Xeon(R) Gold 6230R processors. Two datasets were used for evaluation: *GRCh38 (hg38)* [18] and *Bioinformatics-Dataset* [19], whose details are provided in Table II.

TABLE II
EXPERIMENTAL DATASETS

Dataset	Size (MB)	#Files	Format
GRCh38 (hg38)	3004	24	fasta
Bioinformatics-Dataset	12.7 (single file)	36	fasta

B. Compression Algorithm Performance Comparison

This experiment evaluates the compression performance of the proposed *Gene Sequence Bitmap Compression* (GSBC) algorithm on the *hg38* dataset. The experiment is based on the *Sequence Compression Benchmark Database* (SCB) [20]. To ensure both comprehensiveness and comparability, we selected a variety of general-purpose and domain-specific compression algorithms as baselines. The detailed configuration parameters of these algorithms are listed in Table III.

TABLE III
COMPRESSION ALGORITHM TEST BASELINE

Algorithm	Parameter	Algorithm	Parameter
brieflz [21]	6	leon	2
brotli	6	lizard	45
bzip2	6	lzturbo	31-4t
dlim [22]	–	naf	1
dsrc	4t-m1	pbzip2	1-1t
fastqzf [23]	–	pigz	1-1t
fqzcomp [23]	4	quip	–
gtz	1t-1	xz	1
gzip	5	zpaq [24]	1-1t

The evaluation metrics include: (1) **Compression ratio(CR)** = (compressed file size / original file size) $\times$ 100%; (2) **Compression time** (T_c) and **decompression time** (T_d); (3) **Peak compression memory usage** (M_c) and **peak decompression memory usage** (M_d).

TABLE IV
THE RESULTS OF THE PERFORMANCE COMPARISON OF COMPRESSION ALGORITHMS

Algorithm	CR	T_c (s)	T_d (s)	M_c (MB)	M_d (MB)
GSBC	29%	26.08	12.78	102.58	28.80
brieflz	33%	108.54	11.75	13.12	5.12
brotli	30%	169.08	7.97	74.63	18.31
bzip2	27%	198.72	104.21	5.59	3.39
dlim	21%	183.75	174.21	609.69	65.69
dsrc	24%	20.35	27.36	988.95	619.21
fastqzf	26%	47.04	36.05	10.00	6.44
fqzcomp	21%	89.35	136.10	63.72	65.64
gtz	30%	47.36	33.07	417.29	495.26
gzip	31%	100.70	17.76	1.64	1.63
leon	24%	70.92	84.70	472.26	212.52
lizard	35%	133.34	3.18	38.22	6.78
lzturbo	32%	7.59	2.63	771.65	978.17
naf	23%	9.73	3.69	8.29	64.14
pbzip2	27%	187.86	7.50	7.86	57.25
pigz	35%	35.85	10.71	2.35	2.31
quip	22%	124.14	183.75	340.82	355.27
xz	30%	185.06	51.77	9.84	3.35
zpaq	34%	102.27	4.05	95.65	365.38

In terms of time efficiency, GSBC achieves a compression time of 26.08 s, ranking 4$^{\text{th}}$ overall and outperforming most competitors, with only `lzturbo`, `naf`, and `dsrc` being faster. Its decompression time of 12.78 s ranks 8$^{\text{th}}$, significantly faster than traditional tools such as `bzip2` (104.21 s) and `gzip` (17.76 s). Compared to the top-ranked `dlim` (compression time: 183.75 s) and `quip` (decompression time: 183.75 s), GSBC achieves compression speed improvements of up to 86% and decompression speed improvements of 93%, respectively, indicating superior overall processing efficiency.

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on August 17,2026 at 02:13:33 UTC from IEEE Xplore. Restrictions apply.

In memory usage, GSBC consumes 102.58 MB for compression and 28.80 MB for decompression—only 10.4% and 4.6% of that required by `dsrc`. Relative to parallel-enabled compressors, it delivers up to 43.7× higher compression memory efficiency than `pigz` and 81.2% higher decompression efficiency than `pbzip2`.

Summary:GSBC offers a balanced trade-off between compression/decompression time and memory consumption. This balance enables not only efficient storage of genomic data in compressed form but also facilitates subsequent processing of compressed data, providing a practical foundation for large-scale genomic data management, analysis, and retrieval.

C. Performance Validation of the LFBPBPM

This experiment evaluates the performance of the proposed *Low-Frequency Byte Position-Based Pattern Matching* (LFBPBPM) algorithm under varying pattern lengths. The experiment uses the genomic sequence data from the single file `Cen_25_after.fasta` in the *Bioinformatics-Dataset* as the test text. A pattern set spanning six orders of magnitude in length (from 20 to 10^6 characters) is constructed, where each pattern is a randomly sampled substring from the test text to ensure at least one occurrence in the text.

We compare LFBPBPM against 11 mainstream algorithms. For each test configuration, the total time to search 10 patterns in the test text is measured. Each result is the average of three independent runs, with each run executed separately in the same environment. The evaluation metric is total matching time, where lower values indicate better performance.

Table V shows that LFBPBPM consistently outperforms all competitors across scales. At a pattern length of 1,000,000, it completes in 525 ms, achieving nearly a 4× speedup over Karp–Rabin (2083 ms). Improved algorithms such as EFLPM (834 ms) show enhanced performance relative to traditional methods, but still approximately 60% slower than LFBPBPM. While Zhu–Takaoka performs well at small scales, its runtime reaches 903 ms at this length.

These results are partly explained by the byte frequency distribution after *GSBC* compression of `Cen_25_after.fasta`, where over 80% of byte types occur fewer than 20,000 times. This skew enables LFBPBPM to leverage low-frequency bytes for highly efficient candidate filtering, especially in large-scale searches.

Summary:LFBPBPM demonstrates superior performance across all tested pattern lengths, offering high practical value. The results confirm that the proposed algorithm successfully achieves both sublinear time complexity and millisecond-level responsiveness in pattern matching tasks across multiple orders of magnitude in scale.

D. Experimental Validation of Index Tables

The experimental datasets include all chromosome files from *hg38* and the single file `Cen_25_after.fasta` from the *Bioinformatics-Dataset*. The file sizes of all chromosome sequence files in the dataset are presented in Table VI. Two index construction configurations are evaluated:

- **Full Index Scheme:** Constructs a complete byte-level index for all genomic data.
- **Horizontal Compression Scheme:** Constructs the full index table first, followed by horizontal compression to reduce storage requirements.

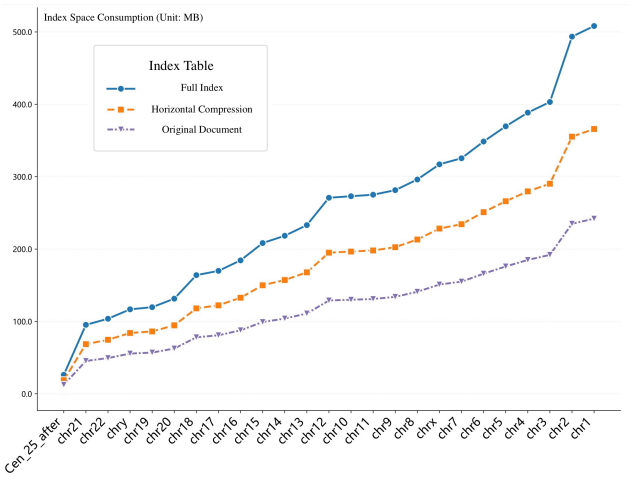

Fig. 6. Comparison of space consumption for building an index table

As shown in Fig. 6, the space consumption of constructing different types of index tables varies significantly across datasets. Taking chr1 as an example, the original file size is 242 MB, whereas constructing a full index table requires 508.2 MB—approximately twice the size of the original file. In contrast, the space required after applying horizontal compression to the index table is reduced to 365.9 MB, representing a reduction of approximately 28%.

These results demonstrate that horizontal compression can substantially reduce the storage footprint of index tables.

VII. CONCLUSION

This work presents a pattern matching framework for compressed genomic sequences with an optimized indexing strategy. The main contributions are as follows:

- **Efficient storage model:** A *Gene Sequence Bitmap Compression* (GSBC) scheme encodes each nucleotide as 2 bits, greatly reducing storage while preserving biological semantics and queryability. It achieves higher efficiency than conventional string formats, especially on large datasets.
- **LFBPBPM algorithm:** A *Low-Frequency Byte Position-Based Pattern Matching* method leverages low-frequency byte sparsity for sublinear-time matching, delivering notable speedups on large genomic data and operating directly on GSBC-compressed sequences without decompression.
- **Index optimization:** A compact index structure with space optimization further improves LFBPBPM efficiency, cutting storage compared with full indexing.

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on August 17,2026 at 02:13:33 UTC from IEEE Xplore. Restrictions apply.

TABLE V
PERFORMANCE COMPARISON OF PATTERN MATCHING ALGORITHMS (UNIT: MS)

Algorithm	20	25	30	35	50	100	500	1000	10000	100000	1000000
BM	1292	2170	1187	1349	1383	1411	1553	1223	1387	1467	1325
Horspool	1279	1176	1167	1016	1058	1200	1156	1129	1035	1189	1247
z Karp-Rabin	3310	2875	2069	1797	1873	2119	1985	2394	2453	2301	2083
Zhu-Takaoka	1173	882	826	758	778	1123	1003	1059	976	1130	903
d-BM	1276	2027	1016	884	921	1327	1209	1540	1610	1365	1461
BF	3777	3452	2959	2564	2664	3389	3285	3090	3001	3352	3185
KMP	2349	1867	1663	1444	1506	1678	1546	1791	1862	1531	1649
FLPM	1061	973	1132	1096	1142	921	1147	915	1322	1421	1371
PAPM	2310	1901	1789	1027	1170	1412	1863	1287	1879	1053	1752
EFLPM	819	758	921	885	947	880	805	920	765	867	834
EPAPM	1769	1552	1497	957	1102	1163	1421	1108	1523	1386	1240
LFBPBPM	503	503	502	597	512	594	501	565	578	592	525

TABLE VI
FILE SIZES OF ALL CHROMOSOME SEQUENCE FILES (UNIT: MB)

Name	Size	Name	Size	Name	Size	Name	Size
chr1	242	chr7	155	chr13	111	chr19	57
chr2	235	chr8	141	chr14	104	chr20	62.6
chr3	192	chr9	134	chr15	99.2	chr21	45.4
chr4	185	chr10	130	chr16	87.8	chr22	49.4
chr5	176	chr11	131	chr17	80.9	chrx	151
chr6	166	chr12	129	chr18	78.1	chry	55.6

Overall, the approach provides an efficient, scalable solution for genomic data storage and analysis, significantly enhancing pattern matching performance.

VIII. ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under Grant No. 2024YFF0729201.

REFERENCES

[1] A. M. Nafea, Y. Wang, D. Wang, A. M. Salama, M. A. Aziz, S. Xu, and Y. Tong, "Application of next-generation sequencing to identify different pathogens," *Frontiers in microbiology*, vol. 14, p. 1329330, 2024.

[2] P. Brlek, L. Bulić, M. Bračić, P. Projić, V. Škaro, N. Shah, P. Shah, and D. Primorac, "Implementing whole genome sequencing (wgs) in clinical practice: advantages, challenges, and future perspectives," *Cells*, vol. 13, no. 6, p. 504, 2024.

[3] W. R. Pearson, "Using the fasta program to search protein and dna sequence databases," in *Computer Analysis of Sequence Data: Part I*. Springer, 1994, pp. 307–331.

[4] P. Deutsch, "Gzip file format specification version 4.3," Tech. Rep., 1996.

[5] J. Gilchrist, "Parallel data compression with bzip2," in *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems*, vol. 16. Citeseer New York, NY, USA, 2004, pp. 559–564.

[6] S. Deorowicz and S. Grabowski, "Compression of dna sequence reads in fastq format," *Bioinformatics*, vol. 27, no. 6, pp. 860–862, 2011.

[7] Y. Xing, G. Li, Z. Wang, B. Feng, Z. Song, and C. Wu, "Gtz: a fast compression and cloud transmission tool optimized for fastq files," *BMC bioinformatics*, vol. 18, no. Suppl 16, p. 549, 2017.

[8] K. Kryukov, M. T. Ueda, S. Nakagawa, and T. Imanishi, "Nucleotide archival format (naf) enables efficient lossless reference-free compression of dna sequences," *Bioinformatics*, vol. 35, no. 19, pp. 3826–3828, 2019.

[9] N. Mansouri Ghiasi, J. Park, H. Mustafa, J. Kim, A. Olgun, A. Gollwitzer, D. Senol Cali, C. Firtina, H. Mao, N. Almadhoun Alserr *et al.*, "Genstore: A high-performance in-storage processing system for genome sequence analysis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 635–654.

[10] P. Leger, H. Fukuda, N. Cardozo, and D. San Martín, "Exploring a self-replication algorithm to flexibly match patterns," *IEEE Access*, vol. 12, pp. 13 553–13 570, 2024.

[11] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne, "Brotli: A general-purpose data compressor," *ACM Transactions on Information Systems (TOIS)*, vol. 37, no. 1, pp. 1–30, 2018.

[12] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa, "A boyer—moore type algorithm for compressed pattern matching," in *Annual Symposium on Combinatorial Pattern Matching*. Springer, 2000, pp. 181–194.

[13] H. Handrizal, A. Budiman, and D. R. Ardani, "Implementation and analysis zhu-takaoka algorithm and knuth-morris-pratt algorithm for dictionary of computer application based on android," *IJISTECH (International Journal of Information System and Technology)*, vol. 1, no. 1, pp. 8–21, 2017.

[14] P. Neamatollahi, M. Hadi, and M. Naghibzadeh, "Simple and efficient pattern matching algorithms for biological sequences," *IEEE Access*, vol. 8, pp. 23 838–23 846, 2020.

[15] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "An alphabet-friendly fm-index," in *International Symposium on String Processing and Information Retrieval*. Springer, 2004, pp. 150–160.

[16] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[17] J. Mathiyalagan, D. Mukherjee *et al.*, "Data structures in artificial intelligence for bioinformatics and computational biology," 2025.

[18] Y. Guo, Y. Dai, H. Yu, S. Zhao, D. C. Samuels, and Y. Shyr, "Improvements and impacts of grch38 human reference on high throughput sequencing data analysis," *Genomics*, vol. 109, no. 2, pp. 83–90, 2017.

[19] "Bioinformatics-dataset," https://github.com/belalahmedhamed/Bioinformatics-Dataset, accessed: 2022-04-21.

[20] K. Kryukov, M. T. Ueda, S. Nakagawa, and T. Imanishi, "Sequence compression benchmark (scb) database—a comprehensive evaluation of reference-free compressors for fasta-formatted sequences," *GigaScience*, vol. 9, no. 7, p. giaa072, 2020.

[21] J. Liu, S. Li, S. Di, X. Liang, K. Zhao, D. Tao, Z. Chen, and F. Cappello, "Improving lossy compression for sz by exploring the best-fit lossless compression techniques," in *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 2021, pp. 2986–2991.

[22] M. H. Mohammed, A. Dutta, T. Bose, S. Chadaram, and S. S. Mande, "Deliminate—a fast and efficient method for loss-less compression of genomic sequences: sequence analysis," *Bioinformatics*, vol. 28, no. 19, pp. 2527–2529, 2012.

[23] J. K. Bonfield and M. V. Mahoney, "Compression of fastq and sam format sequencing data," *PloS one*, vol. 8, no. 3, p. e59190, 2013.

[24] M. Mahoney, "The zpaq compression algorithm," 2015.

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on August 17,2026 at 02:13:33 UTC from IEEE Xplore. Restrictions apply.